# A Pure-Java Group Communication Framework

Carl Cook and Neville Churcher
{c.cook, neville}@cosc.canterbury.ac.nz

Technial Report TR-02/03, July 2003
Department of Computer Science
University of Canterbury,
Christchurch, New Zealand

**Abstract**

This report presents the `caise.messaging` group communication framework—a simple Java-based API developed as the networking component for a collaborative software engineering architecture. The framework is intended to be used as the communication layer for any distributed and/or collaborative systems that have communication requirements beyond simple point-to-point networking, but do not require the services or overheads of fully-featured groupware toolkits.

The `caise.messaging` framework allows groups of remote applications to communicate with each other in the most simple manner as possible. The result is an API that makes every participating application appear local to the calling application, providing communication within the application group by way of conventional method calls.

This report presents an overview of the `caise.messaging` framework, including a background on existing communication technologies, the motivation for a new framework, a summary of the `caise.messaging` architecture, illustrated examples of `caise.messaging`-based tools, and API details.

# Contents

# Chapter 1

# Introduction

This chapter discusses the various existing mechanisms for exchange of data between processes, and then presents a brief description of the `caise.messaging` framework.

## 1.1 Modes of Inter-process Communication

This section presents the different modes of communication that are available when two or more processes wish to exchange data.

### 1.1.1 Asynchronous Communication

It is relatively simple to communicate with remote processes—be it a process on the same machine, or a process on the other side of the Internet. For asynchronous, or one-way, inter-process communication, many well-established technologies may be used: sockets, Rpc [13], Xml [3] messaging, to name just a few.

### 1.1.2 Synchronous Communication

For synchronous communication, where a reply is usually sent in response to a request, the process that issued the request typically just blocks until a reply is received. Request/response synchronous communication is now becoming commonplace. Retrieving resources from a web server is a textbook example, but many other servers now provide programmatic access via a socket connection. Using well defined protocols such as Soap [2], it is possible to communicate synchronously with any Internet-based server.

The capability to communicate with message-parsing servers has gained widespread appeal and application, however the overhead of such systems is relatively high. Xml and other plain-text requests must be constructed and deconstructed by each participating application, and for true event-based communication, every application must also implement a server to receive callback instructions.

### 1.1.3 Middleware-Based Communication

Architectures that are more heavy-weight tend to reduce the programming overhead of message-parsing communication systems. Middleware systems, such as DCom [14] and Corba [9] support remote event-based communication that can be both synchronous or asynchronous, and access to remote clients is enabled via conventional method calls to predefined interfaces. No formation of Xml messages is required, and Xml parsers are also not needed—all communication is performed via method calls, as if all objects are within the same process. Whilst this elegantly hides the aspect of remoteness, the programmatic learning curve for middleware systems is high, and configuration of such systems is often time consuming.

Despite their complications, however, the use of middleware systems is warranted if extensive peer-to-peer or client server communication is required. In the ComAn [7] system for example, it is possible to pass data between any two communicating processes, with the ability to perform user-defined processing at any node within the network.

### 1.1.4 Group Communication

For groupware applications, typically there are many processes all intending to communicate with each other during their lifetime. To implement groupware applications, a large volume of network programming is typically required, in order to facilitate and manage multiple simultaneous one-to-many connections.

In order to reduce the amount of necessary network programming, groupware toolkits gained a degree of popularity. Such toolkits provide components that enabled applications to communication with each other. Groupware toolkits also typically provide configurable floor control policies, multi-user widgets, and other components common to group-based applications.

### 1.1.5 Summary

Whilst simple communication mechanisms exist for point-to-point communication, and groupware toolkits provide elaborate facilities for full-scale multiuser graphical applications, another category of communication must be provided for. Many networked applications such as chat programs, tele-medicine, and collaborative software engineering tools require group-based communication mechanisms, but they do not require specific specific floor control policies, or predefined multiuser widgets.

To this end, this paper introduces an architecture that specifically caters for group-based applications, but only focuses on the establishment of synchronous communication groups, and the delivery of byte-oriented data.

## 1.2 Framework Description

`caise.messaging` is a simple framework for sending streams of data between processes, loosely based on the Java Shared Data Toolkit [4]. An application may register any number of clients, and clients may then subscribe to pre-established meetings. All clients may listen for general events such as the es-

tablishment of new meetings, or specific events such as data being sent across a channel associated with a particular meeting.

### 1.2.1 Accessibility

Applications access the services of `caise.messaging` through a Java Api. The core library is written in pure Java, providing operating system independence. An audio library also exists to provide voice-based communication, but this is currently only available for Linux.

The `caise.messaging` framework is simple to use—it exists as a globally accessible Api. No configuration of the framework is necessary; at runtime the framework configures itself in order to operate. The Api itself is also simple—the main components are Meeting and Channel objects, and data is passed for delivery to all channel Listeners by various 'send' methods.

### 1.2.2 Communication Mechanisms

The `caise.messaging` framework is synchronous—communication is performed via method calls to the Api. Communication may also be event-based. Using the 'meeting' metaphor, clients simply subscribe to meeting channels, and then listen for data being broadcast. Whenever data is broadcasted to listeners of a channel, a callback routine for that client is invoked, informing the client that data is available for inspection.

# Chapter 2

# Background

This chapter provides background information related to the development of the `caise.messaging` framework. We present some similar communication systems, and give brief details on current group applications that use `caise.messaging`.

## 2.1 Related Systems

Many elaborate and well-known groupware systems exist, such as TeamSpace [8], CVW [12], and Lotus Notes [10]. These systems offer support for organisation-wide distribution and editing of documents, along with conference services, and project scheduling facilities. Whilst fully featured and powerful, such systems impose a model of collaboration upon users, and do not support the development of new arbitrary multiuser applications.

The `caise.messaging` framework is in no way an elaborate system. It simply allows groups of users to be established, and then provides an event-based Api for communication within each group. No graphical interfaces are provided, and no mechanisms for the persistence of artifacts exists. Such features must be coded by the application developer.

Systems similar to the `caise.messaging` framework include GroupKit [11]. GroupKit is a Tcl-based multi-user widget set, with implementations for most common operating systems. GroupKit allows groups of users to be established, and also provides some configurable policies to control communication within the groups. GroupKit is very much an 'application replication' environment, where an application is written once, and then multiple instances are used by each group member.

The `caise.messaging` framework differs from GroupKit in three respects. Firstly, `caise.messaging` does not provide any form of multiuser widgets, or any other types of graphical interfaces. Secondly, `caise.messaging` does not provide any floor control mechanisms—users are free to communication whenever they wish to, and it is up to the calling applications to restrict this if required. Finally, `caise.messaging` is not an application replication tool; it is an Api to facilitate interprocess communication. Any number of different applications can join the same communication group and commence communications.

Other systems similar to the `caise.messaging` framework include the Java

Shared Data Toolkit (JSDT) [4]. The `caise.messaging` framework is based upon the JSDT, and shares many concepts, such as client, channel, and meeting objects. The difference between the two frameworks is in `caise.messaging`'s simplicity. `caise.messaging` is a very light-weight framework, requiring little programmatic overhead within calling applications, and very little effort is required to configure `caise.messaging` within a network. The JSDT is certainly more powerful—it supports many options such as networking via an HTTP overlay and group management policies. When designing `caise.messaging` however, we deemed such features to be unnecessary, allowing us to build a simple 'Pure-Java' system with no native library dependencies.

JavaGroups [1] is also another group messaging framework for the Java programming language. JavaGroups again provides an API for application development, and also supports the metaphor of clients and channels. Many similarities exist between JavaGroups and `caise.messaging`, but again, `caise.messaging` is a much simpler framework.

## 2.2 Current Use

The `caise.messaging` framework has been used in several research applications. In this section we present two such systems.

### 2.2.1 CAISE

As mentioned in the abstract of this report, the `caise.messaging` framework is the networking component for a collaborative software engineering architecture [5, 6]. This overall architecture is called CAISE (Collaborative Architecture for Iterative Software Development), and consists of many different software tools that may be used for collaborative programming and system design.

The overall CAISE architecture is based upon a central server for all software projects. The server maintains the code repository for all software artifacts, and also controls the communication between users and applications that are currently working on each software project. One of the key underlying components of such a system was a decoupled communication mechanism, and this was the motivational factor behind the development of `caise.messaging`.

The `caise.messaging` framework allows each participating software engineering tool to connect to the central server. In the CAISE architecture, each tool is treated as a client of a project-specific channel, and all messages and data for each tool are sent directly between the server and the client via the `caise.messaging` API. Additionally, the CAISE architecture allows communication directly between a pair or group of participating software tools; again the `caise.messaging` framework is invoked to facilitate such communication.

At present, the CAISE architecture supports several system design tools and code editors. Each character by character code update and each cursor or mouse movement for every user is broadcasted within the CAISE architecture. With many users working on the same project simultaneously throughout the network, the underlying `caise.messaging` framework keeps up, suggesting that most distributed applications will find `caise.messaging` suitable in terms of performance.

### 2.2.2 A Distributed Extreme Programming Environment

The `caise.messaging` framework is also being used as the underlying group communication mechanism for a distributed eXtreme Programming (XP) environment. The environment, named Dxp [15], maintains user stories and task iterations for any number of projects. Every programmer uses a development environment tailored for the manipulation of XP artifacts, and `caise.messaging` provides the underlying messaging service to enable distributed communication.

# Chapter 3

# Design and Implementation

This chapter outlines the architectural design of the `caise.messaging` framework. Major implementation aspects are also presented.

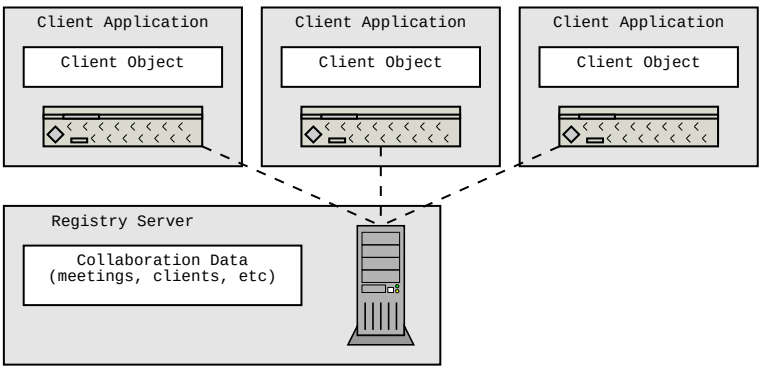
## 3.1 Architecture

Central to the `caise.messaging` framework is a registry server process that maintains the state of all collaborating clients. The application programmer needs only to be aware of the location of the `caise.messaging` registry server; once running, the server process requires no further intervention. End users of `caise.messaging`-based collaborative tools need not be aware of such a server.

The framework allows Java applications to establish channels for communication. Upon establishment, applications may begin listening to such a channel. Applications listening on the same established channel effectively form a group, in which packets of any type of data can be broadcasted or addressed to specific group members in an event-based manner.

### 3.1.1 General Schema

The following diagram presents the general architecture of `caise.messaging`:




Each application creates and registers a `caise.messaging` Client object with the `caise.messaging` registry server via an Api call. At this point, all internal

mappings between the `caise.messaging` registry server and the application are made. Calls can then be made to the Api to join or create new meetings, and to send data to other listening clients. Upon close-down, the application may request to remove all associated clients from established meetings.

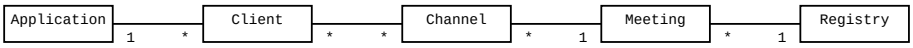
### 3.1.2 Mandatory Components

As mentioned previously, the `caise.messaging` registry server must be running on one workstation within the network. Section 4.1 provides details on how to start the registry server. Once the `caise.messaging` server process is active, applications may call upon the `caise.messaging` Api.

### 3.1.3 Key Classes

Several classes are utilised by all `caise.messaging` collaborative applications. The first class to be obtained is the `Registry` class, which represents a local copy of the running `caise.messaging` registry server. This class allows the creation of `Client` objects, which are basically unique identifiers for the purposes of meeting subscriptions and the addressing of data streams.

Once the local registry object has been obtained, `Meeting` objects can also be obtained, either by naming a pre-existing meeting or creating a new one. Similarly, once a meeting object has been obtained, `Channel` objects can also be created or located. The following diagram shows the cardinality between the key classes:




As the diagram above shows, a meeting can have any number of channels. This is useful when a collaborative application has several independent streams of data to manage, such as a text channel, a voice channel, and an application-specific data channel. Therefore, clients must select a specific channel when sending data, rather than just specifying the appropriate meeting.

Several other classes are also core to `caise.messaging`, and they fall under two categories: Data classes and Event classes. A `Data` class holds a stream of data in the forms of a byte array or a string. To send a data object on a specific channel, an `Envelope` object is created, with the data object attached. Whenever data is sent a new event is raised to all listening clients; clients may inspect this data if they choose to do so. Events are also generated when channels or meetings are established, subscribed to, or terminated; to listen to such events clients must add a `Listener` to the local registry object.

# Chapter 4

# Example Usage

This chapter details a few example applications that call upon the `caise.messaging` framework. As for all `caise.messaging` applications, ensure that the `caise.messaging` registry server is already running.

## 4.1 Preliminaries

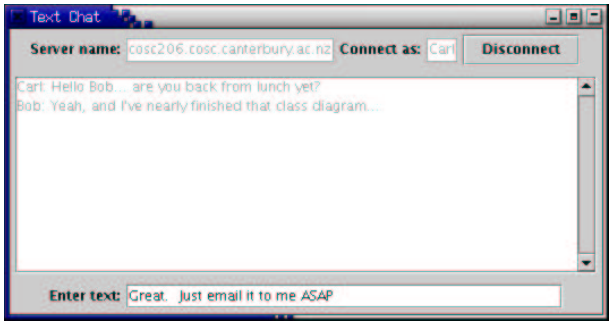
In order to run any of the example applications, or any other `caise.messaging` based application, the CAISE registry server must be running on one machine accessible within the network.

To start the CAISE registry server, the command `java caise.messaging.Registry` should be issued. The program will shortly terminate, leaving the registry server running in its own separate process. All output and error messages will be written to log-files.

An alternative way to execute the registry server is via the `caise.messaging` API. An application can call the static method `Registry.run(boolean logFiles)` which will also start the server. This method blocks until the server is shut down, and so the caller is required to invoke this method via a new thread if control within the current thread is required. The boolean argument `log-files` specifies whether or not log-files should be written. If this value is false, all error and output messages will be written to standard out.

## 4.2 A Text-Based Chat Client

The file `src/testapps/messaging/TextChat.java` contains the code for a well commented text-based chat program. The file contains three classes, with the one being described herein called `TextChatImpl`. After starting the `caise.messaging` registry server on an arbitrary machine, the TextChat application may be run on any number of machines. Upon application startup, the name of the machine that is running the `caise.messaging` registry server should be specified in the "server name" textbox. The application GUI is presented in the following figure:




The first point to note is that this class implements the `CAISEListener` interface, as it will receive callbacks from the `caise.messaging` registry server via the `update()` method. Every time the update method is called with an event of type `CAISEEvent.DATA_SENT`, the data associated with the event is placed into a collection. This collection must be synchronised as the update method is invoked by a thread separate to the main application thread.

Accordingly, the next method in the `TextChatImpl` class is the main application loop; this method retrieves data from the local collection. Again, access to the collection must be in a synchronised block to prevent potential concurrency issues. If data exists, it will be in the form of an `Envelope`; it is casted as such when it is removed from the collection. The envelope is then inspected with the data extracted. The method `getSender()` allows us to write the name of the person chatting to the application GUI, and similarly, the data string from the `data` object is also displayed.

To establish a new meeting and channel, the method `setUpMeeting()` is provided. This method creates a reference to the local registry object, passing the name of the new client in the form of an arbitrary string. Internally, a new client object is established, and a reference to this object is then obtained by subsequently querying the local registry object. Next, a meeting and a channel are established. The meeting and channel names are hardcoded—this is not a problem as only this application needs to be aware of this meeting. In the case where this is the first TextChat application to be executed, a new meeting and channel are created. If a meeting and channel already exist (because instances of the application are already running), the meeting and channel are simply joined to the client instead. The final major step in initiating a `caise.messaging` session is to add a listener to either the entire `caise.messaging` registry or just a channel. In the case of this application, we are only interested in data being sent via the newly established channel, so a channel listener is added in the form of the `TextChatImpl` class.

As an extension of a simple chat program, a `TalkButton` is also added to the application GUI, with the button associated with the current meeting. This is not core to the implementation of the chat program, but it does demonstrate a meeting with multiple channels. The TalkButton component is explained in section 4.4.

The operations of the next method, `closeDownMeeting()`, basically reverses those of the method `setUpMeeting()`. The client object is removed from the channel and meeting, and if this instance of the application is the last instance to leave, the channel and meeting are deleted.

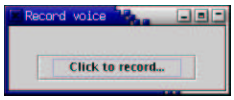
The final method of importance in this application is `enterPressed()`; this method handles new messages that are to be distributed to all other clients associated with the TextChat channel. Given the message in String format, a new `Data` object is created and initialised. Next, an `Envelope` object is created via the Client Api, with the data object attached. The data is then sent to all listeners on the channel via the method `channel.sendToAll()`.

## 4.3 A Minimal `caise.messaging` Program Pair

A trivial example of two different programs communicating over the same channel may be found in the directory `src/testapps`. The program `FirstPerson.java` establishes a new meeting and channel, and then waits for data to be sent back. The program `SecondPerson.java` joins the channel and then sends three packets back to the first application. The fourth packet contains the string "quit", which the first application recognises and quits accordingly. At this point, the second application also closes down. Please refer the the source code comments for further details.

## 4.4 An Audio Player

As an additional simple example program, the program `src/testapps/messaging/TalkFrame.java` demonstrates the use of the `caise.util.audio.TalkButton` Swing component[1]. This component takes the form of a `JButton`, but has the added feature of recording voice when the button is held, and playing back the voice to all other listeners when the button is released. To enable the voice operations, the class `caise.util.audio.AudioMixer` is used—at present this class is only available on the Linux platform due to its use of native libraries. The application GUI is presented in the following figure:




An interesting feature of the TalkButton component is the internal establishment of a meeting, channel, and a channel listener. In a manner similar to that of the TextChat application, the first running instance of the TextButton will create a new meeting and channel, otherwise the client object is simply joined to the hard-coded meeting and channel. The closedown routine is also similar to that of the TextChat application. As no other party requires access to TextButton's meeting and channel objects, the meeting and channel names are hard-coded as private constants.

As the TalkButton component also implements the `CAISEListener` interface, all applications that use a TalkButton also receive packets of data in the form of voice messages. Upon the arrival of data events, the TalkButton casts them as binary data and plays them via the AudioMixer library.

Since the TalkButton component internally handles meeting and channel setups and teardowns, recording of voice messages, distribution to all other

[1] execute the program via the command `java testapps.FirstPerson SERVER` where `SERVER` is the name of the machine that is running the `caise.messaging` registry server

channel listeners, and playback of messages from all other channel members, the TalkFrame example application has very little significant source code. No calls to the CAISE core classes are required; the only essential step is to construct a TalkButton object with the name of the `caise.messaging` server (which is passed as a command line argument in this example).

# Chapter 5

# The Application Programmer Interface

This chapter provides an overview of the most significant classes and interfaces. For a detailed description of each class and interface, please refer to the javadoc Api listing of the `caise.messaging` framework.

## 5.1 The caise.messaging Package

The `caise.messaging` package contains the core `caise.messaging` classes for collaborative applications.

### 5.1.1 The Registry Class

The Registry class represents the remote Caise registry server. This class provides the static method getRegistry(), which is the only way to gain access to the registry services.

| | | |
|---|---|---|
| void | addListener( CAISEListener listener) | Add a listener to the remote registry |
| void | addMeeting( String name, Client creator) | Add a new meeting to the registry |
| void | deleteClient( Client client) | Remove a client from the registry |
| void | deleteMeeting( Meeting meeting) | Delete a meeting from the registry |
| Client | getClient( String name) | Retrieve a reference to a client from the registry |
| Map | getClients() | Get the list of clients currently associated with the registry |
| Meeting | getMeeting( String name) | Retrieve a meeting from the registry |
| Map | getMeetings() | Get the list of meetings currently associated with the registry |
| static Registry | getRegistry( String hostName, String client) | Returns the Registry object for the given server. |
| void | leave( ) | Removes all resources from the registry associated with the current client |
| static void | main( String[] args) | Called to start the CAISE registry server process |
| void | removeListener( CAISEListener listener) | Remove a listener from the remote registry |
| static void | run() | Called to start the CAISE server from within another application |
| void | shutdown() | Shut down the remote registry |

### 5.1.2 The Client Class

Represents a client of the `caise.messaging` framework. A client is owned by a collaborative application, and may join any number of meetings and channels. Clients may send and receive packets of data.

| | | |
|---|---|---|
| Envelope | createEnvelope(Data data, String receiver) | Create a new envelope to be sent from this client, initialised with data and the name of the receiving client |
| Map | getChannels() | Return the list of channels that this client is currently subscribed to |
| Map | getMeetings() | Return the list of all meetings that this client is currently subscribed to |

### 5.1.3 The Meeting Class

The meeting class represents an meeting facilitated by a `caise.messaging` registry server. A meeting can have any number of clients associated with it. A meeting can own any number of channels. A new meeting object is obtained by a call to Registry.addMeeting(String, Client).

| | | |
|---|---|---|
| void | addChannel(String name, Client creator) | Adds a new channel to the Meeting |
| Channel | getChannel(String name) | Returns a reference to the named channel |
| Map | getClients() | Returns a list of all the clients that are currently associated (subscribed) with this channel |
| void | join(Client client) | Adds a client to the current meeting. |
| void | leave(Client client) | Removes a client from a previously joint meeting |

### 5.1.4 The Channel Class

Represents a channel that clients may send and receive data on. Channels belong to a specific meeting, and any client that has joint a channel's owning meeting may join that channel.

| | | |
|---|---|---|
| void | addListener(CAISEListener listener) Add a listener for events associated with this channel | |
| Map | getClients() | Get a list of all the clients currently joint to this channel |
| Meeting | getMeeting() | Return a reference to the owning meeting |
| Client | getOwner() | Return a reference to the client who created this channel |
| void | join(Client client) | Join a client to the channel |
| void | leave(Client client) | Remove a client from the channel |
| Data | receive(Client client) | Blocking method for receipt of data. |
| void | send(Envelope env) | Send a packet of data on this channel to a specific client. |
| void | sendToAll(Envelope env) | Send a packet of data to all clients listening on this channel. |
| void | sendToOthers(Envelope env) | Send a packet of data to all clients (except the sender) listening on this channel. |

### 5.1.5 The Data Class

The Data class represents a packet of data, in either String or byte array format

| | | |
|---|---|---|
| static Data | createData(byte[] data) | Create a new data packet, populated with an array of bytes |
| static Data | createData(String data) | Create a new data packet, populated with a String |
| byte[] | getDataBytes() | Retrieve the data from a packet in the form of an array of bytes |
| String | getDataString() | Retrieve the data from a packet in the form of a String |
| long | getLength() | Returns the length of the data (in bytes) |

### 5.1.6 The Envelope Class

An Envelope carries a Data object for transport to a Client or Clients. A new Envelope object is created via a call to Client.createEnvelope(), and it may then

be populated with a Data packet, and a receiver in the form of a Client. An Envelope may then be sent on a Channel via the method Channel.send(Envelope), or upon receipt of an Envelope, the Data may be inspected and used.

| | | |
|---|---|---|
| Channel | getChannel() | Get the channel that the Envelope will be sent on |
| Data | getData() | Get the Data packet for this Envelope |
| Client | getReceiver() | Get the receiver for this Envelope |
| Client | getSender() | Get the sender of the Envelope |
| void | setData(Data data) | Set the Data packet for this Envelope |
| void | setReceiver(Client receiver) | Set the receiver for this Envelope |

### 5.1.7 The CAISEListener Interface

Observer interface for the Registry or a Channel. Whenever an event occurs on the observed object, the update(Object, CAISEEvent) method is invoked on the implementing listener

| | | |
|---|---|---|
| void | update(Object src, CAISEEvent evt) | The callback method for implementors of the CAISEListener interface. If the event was thrown by the Registry, the source may be the channel, meeting, or client that caused the event. If the event was thrown because data was sent, the source will be the data packet in the form of an Envelope. |

### 5.1.8 The CAISEEvent Class

Typesafe enumeration for CAISEEvents. Given an event object, the equals(CAISEEvent) method will determine if it matches one of the constant CAISEEvent objects.

| | | |
|---|---|---|
| boolean | equals(CAISEEvent e) | Determines if the given CAISEEvent object equals the current object. |

### 5.1.9 The CAISEException Class

Generic exception class, normally extended for more specific Caise Exceptions, such as NoSuchClientException, etc

## 5.2 The caise.utils Package

The `caise.utils` package contains helper classes that are not considered core to the `caise.messaging` framework.

### 5.2.1 The Notify Class

Utility class to print out debug and error messages

| | | |
|---|---|---|
| static void | debug(String args) | If the system property "caise.debug" is set to "true", the message args will be printed to standard out |
| static void | error(String args) | The message args will be printed to standard error |
| static void | warn(String args) | The message args will be printed to standard out |

## 5.3 The caise.utils.audio Package

The `caise.utils.audio` package contains classes that provide audio functionality for collaborative applications. These classes are not considered core to the `caise.messaging` framework.

### 5.3.1 The AudioMixer Class

Utility class for recording streams of voice data and playing them back

| | | |
|---|---|---|
| | AudioMixer(boolean duplex) | Create a new mixer object. |
| void | close() | Close the sound device (if left open by full duplex construction) |
| void | play(byte[] audioBytes) | Plays out the audio data, as returned by recordStop() |
| void | recordStart() | Start recording data (non blocking method - will return immediately) |
| byte[] | recordStop() | End recording data (returns data stream as an array of bytes) |

## 5.4 The caise.client.widgets Package

The `caise.client.widgets` package provides multi-user widgets for the CAISE collaborative development environment. For the purposes of `caise.messaging`, however, the only relevant class of concern is the TalkButton class.

### 5.4.1 The TalkButton Class

Extension of a Swing JButton that provides collaborative audio facilities. When the button is held down, it records a stream of audio data. When the button is released, it sends this data to all other clients connected on the same CAISE registry server that use a TalkButton. When voice packets from other TalkButtons arrive, they are played out.

| | | |
|---|---|---|
| | TalkButton(Meeting meeting) | Creates a new channel for the TalkButton, based on the given meeting. |
| | TalkButton(String serverName) | Creates a new button object given the name of the CAISE server A new meeting and channel will be created, along with a new anonymous client |
| void | detach() | close down the meeting and channel created when this button was constructed |

# Chapter 6

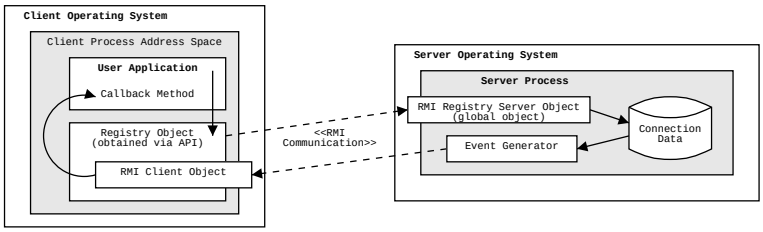
# Internal Mechanisms of the `caise.messaging` Api

This chapter provides details about the finer aspects of the `caise.messaging` framework, including how remote communication is achieved, and how resources are allocated and deallocated. Instructions are also given relating to the execution of new `caise.messaging` based applications.

## 6.1 Application Startup

Upon the client's initial call to Registry.getRegistry(), a local copy of the registry server is created within the client's address space. This allows the local registry object to communicate with the remote registry server, whilst appearing local to the calling client.

When the local registry object is obtained from within the client application, a local callback object is created internally and added to the local Rmi registry. The local registry object then connects to the remote registry server, and immediately the remote registry server binds back to the local callback object. This way, both calls to and from the client are possible, providing a mechanism for event callbacks and delivery of user data, as shown in the following figure:




## 6.2 Notes on Usage

The following points provided guidelines and pointers for the creation of `caise.messaging`-based applications:

- When Registry.getRegistry(hostName, clientName) is called, the `rmiregistry` process on the calling workstation is invoked. For all applications that require the RMI API, normally the user must invoke the RMI registry. For the `caise.messaging` framework, however, this process is automatically invoked at application startup.
- When Registry.deleteClient() is called, all internal associations are removed. Any established meetings and channels remain.
- When a meeting is deleted, all associated channels are deleted. All associated clients (to both the meeting and the channel) remain.
- The method Registry.leave() removes all resources from the registry related to the client that made the call. This method should be called for every prior call to Registry.getRegistry(hostName, clientName). Failure to call Registry.leave() prior to reconnecting to the registry can cause subsequent unpredictable behaviour.
- It is recommended to keep the update method of all CAISEListener implementations to a minimum. A synchronized collection to store events in should also be used (please refer to the sample applications). Simply move the messages to the collection when the update method is called, and use the main application thread to remove the messages from the collection for processing. Again, the removal of the messages must be synchronised.

## 6.3 Execution of User-Defined `caise.messaging` Applications

### 6.3.1 Constraints

As mentioned in section 4.1, the CAISE registry server must be running on a machine visible within the network. This machine must be specified whenever a call to `Registry.getRegistry(hostName, clientName)` is made.

Additionally, the full classpath must be specified as JVM options when running any `caise.messaging` based application. This is because `caise.messaging` parses the classpath, and supplies this to the RMI libraries (which in turn attempt to locate the `caise.messaging` RMI stubs). Therefore, the classpath setting must be explicit, and use an absolute reference to the classes directory (for example `/home/users/carl/myapp/classes`).

### 6.3.2 Options

The following optional definitions can be made when invoking the Java virtual machine:

- For debugging information at runtime, make the following definition: `-Dcaise.debug=true`
- To specify the location for additional native libraries, make the following definition: `-Djava.library.path=lib`

## 6.4 Rmi Policy Files

There is no Rmi policy file for the `caise.messaging` framework. Even though `caise.messaging` relies on Rmi, all Rmi settings are configured dynamically. The following Rmi settings can however be overridden:

- `-Djava.rmi.server.codebase=file://[classpath]`
- `-Djgk.rmiregistry=/usr/bin/rmiregistry`

# Chapter 7

# Building the Classes

This chapter provides details on how to build the `caise.messaging` framework and any new `caise.messaging` based applications. Some troubleshooting advice is also provided.

## 7.1 Building the Core Libraries

The Makefile (as supplied within the distribution) is the easiest way to build the core `caise.messaging` framework. The command `make all` will make the server classes, the native libraries, the user documentation, the API documentation, and the test client applications. The command `make all-server` will make all the server classes, RMI stubs, and native libraries, whilst the command `make server` will make just the core server classes, excluding the native libraries and RMI stubs. The command `make all-clients` will build only the client applications. Please refer to the Makefile for more details on how the classes are built.

## 7.2 Building New Applications

No additional steps are required to build new `caise.messaging` based applications. The only requirement, as with all Java libraries, is that the `caise.messaging` classes are within the Java compiler's classpath.

## 7.3 Bugs and Feedback

Please email `c.cook@cosc.canterbury.ac.nz` with any bug reports or general items of feedback. Whilst the sample applications have been tested, there are no doubt many bugs still present in the system. There may also be (hopefully minor) design flaws, as well as mechanisms that have not yet been implemented.

One known bug is that on client shutdown, the exception `caise.messaging.InconsistentStateException` is often thrown. This is just a concurrency issue that I have not yet resolved, and doesn't compromise the system's behaviour.

Additionally, it is not yet possible to run `caise.messaging` from a Jar file. This is because `caise.messaging` inspects the class path of the calling

JVM to determine the location of the RMI stub files. In a future version of `caise.messaging`, it will however be possible to include the `caise.messaging` classes within a Jar file.

## 7.4 Troubleshooting

- Occasionally, a `java.rmi.ConnectException` or `java.net.ConnectionRefused` exception may occur upon startup after recompiling the server classes. If this happens, the `rmiregistry` process should be terminated (root access might be required to do this). This will force an update of the running rmi stub/skeleton cache.
- Occasionally, RMI will crash the Java virtual machine. This typically happens when RMI is flooded with large packets of data, but this has also occurred at least once so far when a single, small packet has been sent. There are many open bugs of this nature with Java... all we can do is wait for such bugs to be fixed.

# Appendix A

# Availability

The latest version of the `caise.messaging` framework is available (in tarfile format) from the author. This includes all documentation, source code, and test applications. Additionally, the entire CAISE architecture, and suit of collaborative software development tools, is also available. Please contact the author (`c.cook@cosc.canterbury.ac.nz`) for the source code. The source code is released for unlimited use and development for research purposes.

# Bibliography


[1] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java, September 1998. Cornell University.

[2] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol 1.1. Technical report, W3C Consortium, May 2000. See `http://www.w3.org/TR/SOAP`

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language 1.0 Second Edition. Technical report, W3C Consortium, October 2000. See `http://www.w3.org/TR/REC-xml`

[4] R. Burridge. *Java Shared Data Toolkit User Guide*. Sun Microsystems, October 1999. Online document, Available from *java.sun.com/products*.

[5] C. Cook. *Towards Computer-Supported Collaborative Software Engineering*. PhD thesis, University of Canterbury, Christchurch, New Zealand, December 2004. Work in Progress.

[6] C. Cook and N. Churcher. An Extensible Framework for Collaborative Software Engineering. In *10th Asia-Pacific Software Engineering Conference (APSEC)*, Chiangmai, Thailand, December 2003.

[7] C. Cook, K. Pawlikowski, and H. Sirisena. ComAN: A Multiple-Language Active Network Architecture Enabled via Middleware. In *5th International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, USA, July 2002. IEEE.

[8] L. Fuchs, S. E. Poltrock, and I. Wetzel. TeamSpace: An Environment for Team Articulation Work and Virtual Meetings. In DEXA *Workshop*, pages 527–531, 2001.

[9] O. M. Group. The Common Object Request Broker: Architecture and Specification. Document 96-03-04, OMG, Framingham, MA and Reading, Berkshire, UK, June 1995.

[10] B. Reinwald and C. Mohan. Structured Workflow Management with Lotus Notes Release 4. In *Proceedings of the 41st IEEE Computer Society International Conference (CompCon)*, pages 451–457, Santa Clara, California, February 1996.

[11] M. Roseman and S. Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[12] P. Spellman, J. Mosier, L. Deus, and J. Carlson. Collaborative Virtual Workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–203, Phoenix, AZ, November 1997. ACM Press, NY.

[13] I. Sun Microsystems. *Remote Procedure Call Protocol Specification*. Internet Standard, April 1998. RFC 1050.

[14] Y.-M. Wang and P.-Y. E. Chung. Exploring Customization of Distributed Systems using Com. In *Concurrency Magazine*. IEEE, July 1998.

[15] M. M. Williams. Distributed Extreme Programming: Extending the Frontier of the Extreme Programming Process. Master's thesis, University of Canterbury, Christchurch, New Zealand, July 2003. Work in Progress.